Versionando con Git y Github - Parte 2

Enviado por JStitch el Mar, 12/07/2011 - 10:42.

o Cómo Introducirse al Software Libre siendo Programador

Esta es la parte 2 de una serie de posts que estoy realizando para hablar tanto de versionadores en general (y de Git y el servicio GitHub en particular) como del hecho de que para participar en un proyecto de software (de cualquier índole, pero en especial los de software libre) un control eficiente pero a la vez sencillo de las versiones del código es importantísimo.

En la parte 1 introduzco los conceptos de versionadores, esquematizo el proceso que en general se sigue para utilizar un sistema así, hablo de la historia de los versionadores y termino diferenciando los versionadores centralizados de los distribuidos.

En esta parte hablaré de Git, un sistema de control de versiones distribuido bastante popular.

Si ya conoces sobre Git y te interesa pasar directamente a la parte de GitHub, puedes ir a la parte 3.

Git

Este post se concentra en introducir el sistema de control de versiones (version control system, VCS) Git.

Para explicar Git y sus conceptos como VCS, utilizo la interfaz más común del sistema: la línea de comandos. Por ello, este post está plagado de comandos de la mano de los conceptos explicados.

Mi recomendación para entender Git es entender los conceptos con los comandos asociados, y después si así se desea, buscar alguna interfaz diferente más acorde a los gustos personales. Al final de este artículo coloco unos links con algunas interfaces alternativas para Git. Lo que sí debe quedar claro es que este post es una mera introducción a Git, no pretende ser una guía exhaustiva del sistema. La idea es darlo a conocer, promover su uso, hacer notar su importancia en proyectos de software libre, y dejar la carta abierta para que el interesado investigue más y se informe de todos los detalles que aquí no se mencionan.

Para facilitar la lectura de este artículo y que a su vez pueda servir como una pequeña referencia introductoria a Git, coloco aquí ligas internas a las distintas secciones del mismo, a manera de índice:

- Breve historia de Git
- El modo Git
- Generando un repositorio
  - 1 - Inicializando un directorio como repositorio Git
  - 2 - Copiando un repositorio Git
- Usando el repositorio local
  - git add
  - git status
  - git diff
  - git commit
- Branches y Merges
  - git branch / checkout
  - git merge
  - Conflictos
  - git log
  - git tag
- Interactuando con un repositorio remoto
  - git remote

- git fetch-merge / pull
- git push
- git request-pull

- Conclusión
- Interfaces para Git
- Para saber más...

Breve historia de Git

Git surgió como un VCS de la mano de Linus Torvalds, el creador del kernel Linux, para administrar precisamente el código de su proyecto estrella.

Inicialmente el kernel de Linux era administrado con BitKeeper, un VCS de código cerrado que, paradójicamente para muchos, permitía administrar uno de los productos estelares en el ambiente del software libre. Como es propio de él, Torvalds no cedía a presiones basadas en preferencias y filosofías y continuó con BitKeeper hasta que este proyecto lanzó restricciones que no permitían un uso libre de los proyectos hosteados con ellos. Así pues como buen hacker, Linus Torvalds comenzó a escribir Git para llenar el hueco dejado por BitKeeper de un VCS distribuido, elegante pero sobre todo eficiente para usar con Linux. Si toda esta anécdota es o no una lección sobre filosofía del uso de software libre, quede a criterio del lector.

Desde su lanzamiento, el 6 de abril del 2005, Git se ha convertido en uno de los principales VCS distribuidos, sobre todo (pero no exclusivamente) en el mundo del software libre. Algunos de los proyectos que, a día de hoy, utilizan Git como VCS son:

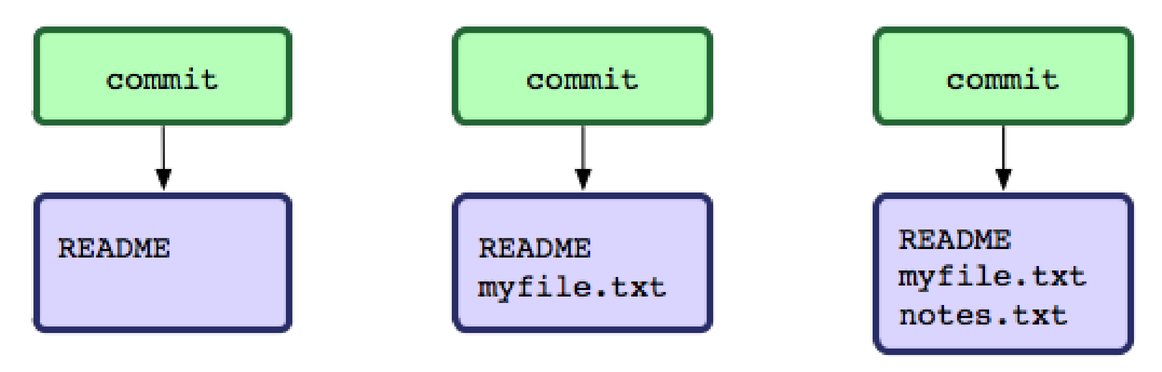
- el kernel de Linux
- el proyecto Android
- el CMS Drupal
- el entorno de escritorio Gnome
- el lenguaje de programación Perl
- el manejador de ventanas Openbox
- Wine, los servicios de compatibilidad Linux-Windows
- las coreutils del proyecto GNU
- X.Org, el servidor gráfico para entornos Unix
- Qt, el framework de aplicaciones gráficas
- Samba, la implementación libre del protocolo SMB para compartir archivos con sistemas Windows
- y un largo etcétera...

El modo Git

Muy al contrario de como se maneja un VCS tradicional (como Subversion), Git maneja los repositorios, y los conceptos mismos de un VCS de una manera particular.

Mientras que para Subversion, el control de versiones se hace archivo por archivo, sobre cada directorio que integra un proyecto, para Git el control de versiones se hace sobre los distintos 'snapshots' que se tomen de todo el proyecto.

La diferencia radica en que para sistemas como Subversion, cada versión del proyecto incluye la versión de cada uno de los archivos del mismo. Mientras tanto para Git, cada versión del proyecto incluye solamente un manifiesto con las diferencias de cada archivo, es decir de cómo se ve el proyecto en su totalidad en determinado momento. Entendiendo Git así, será fácil adaptarse a su modo de funcionamiento, e incluso salir de usar un VCS centralizado y comenzar a usar la genial idea que representan los VCS distribuidos.



De manera muy general, el siguiente esquema ilustra cada parte del proceso que se usa para versionar con Git:

Proceso de versionado con Git

Generando un repositorio

El primer paso para utilizar Git es tener un repositorio. El repositorio Git se representa por un directorio especial llamado **.git** y que reside en el directorio raíz del proyecto mismo. A diferencia de SVN, Git sólo genera un único directorio para todo el repositorio, en donde almacena toda la información del mismo (versiones, historial, etc.); si se recuerda SVN, éste almacena un directorio .svn dentro de cada subdirectorio del proyecto versionado, almacenando en el mismo esa misma información (con otro formato) para cada directorio y por lo tanto cada archivo del proyecto.

Hay dos maneras de generar un repositorio para trabajar con Git sobre un proyecto:

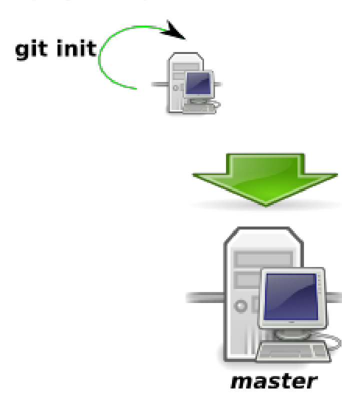
1 - Inicializando un directorio como repositorio Git

Suponiendo que se tiene un directorio en el que reside el proyecto a versionar, el comando

git init

crea el repositorio Git para el proyecto dado. Esta generación del repositorio es completamente local a la máquina y el directorio donde reside el proyecto. Ninguna operación se realizó comunicándose con algún servidor ni nada por el estilo.

```
$ cd proyecto
$ git init
Initialized empty Git repository in /path_to_proyecto/proyecto/.git/
```



2 - Copiando un repositorio Git

Ahora supongamos que deseamos colaborar en algún proyecto (o simplemente queremos obtener su código fuente para compilarlo y usarlo, o para mirarlo -- y admirarlo :). Para esto, lo primero que hay que hacer es obtener el código fuente. Si los administradores del proyecto son listos y utilizan Git para el mismo, lo que debemos hacer es copiar o *clonar* el repositorio origen del proyecto para así tenerlo como un repositorio completamente local sobre el cual poder trabajar.

Los repositorios git tienen una URL, y con ella se utiliza el comando

git clone [url]

para clonar el repositorio de origen remoto a un repositorio completamente local sobre el cual poder trabajar. Por ejemplo:

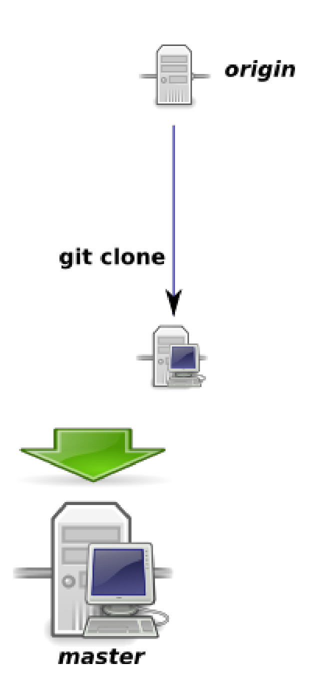
```
$ git clone git://github.com/jstitch/masterm.git
Cloning into masterm...
remote: Counting objects: 36, done.
remote: Compressing objects: 100% (35/35), done.
remote: Total 36 (delta 14), reused 0 (delta 0)
Receiving objects: 100% (36/36), 49.14 KiB, done.
Resolving deltas: 100% (14/14), done.
```

clonará el repositorio del proyecto que reside en `git://github.com/jstitch/masterm.git` al directorio `masterm` local:

```
$ cd masterm
$ ls
BUGS curstextlib.h INSTALL languages.h Makefile masterm.h TODO
cursors.h HISTORY intl/ LICENSE masterm.c README utils.h
```

Si se observa el contenido de los archivos ocultos de este directorio, se podrá ver que efectivamente se tiene un repositorio Git en él:

```
$ ls -a
./ BUGS curstextlib.h HISTORY intl/ LICENSE masterm.c README utils.h
../ cursors.h .git/ INSTALL languages.h Makefile masterm.h TODO
```



Independientemente del método utilizado, antes de comenzar a trabajar, y si no se ha hecho aún, es necesario indicarle a Git los datos que utilizará el sistema para saber qué usuario será el responsable de los cambios hechos (de otra manera no tendría sentido el usar un VCS, sin tener a quién ~~echarle la culpa~~ darle gracias por los cambios hechos al código). Esto se logra con los siguientes comandos:

```
$ git config --global user.name 'Tu nombre'
$ git config --global user.email tu@dominio.com
```

Usando el repositorio local

El funcionamiento básico de Git consiste en trabajo local, trabajo local y trabajo local: modificando archivos, generando branches, haciendo merges con ellos, agregando los archivos con cambios que se deseen versionar, versionándolos y así sucesivamente. Solamente cuando ya se tiene un conjunto de código y cambios *hechos y probados* se procede a mandarlos al repositorio origen desde el cuál se clonó (o a solicitar que se integren los cambios hechos al mismo en caso de no tener los permisos adecuados).

En resumen, se utiliza `git add` para agregar los cambios a los archivos que se desea que Git tome en cuenta para la siguiente versión del código, `git status` y `git diff` para observar los cambios puntuales que se realizarán para la siguiente versión y `git commit` para almacenar dicha versión en el historial del repositorio. Este es el flujo principal que se sigue al trabajar con Git, y hay que destacar que todo se hace de manera local, sin interacción con repositorios remotos: recuérdese que se está trabajando sobre un repositorio local y que precisamente éste es el sentido de los VCS distribuidos.

`git add`

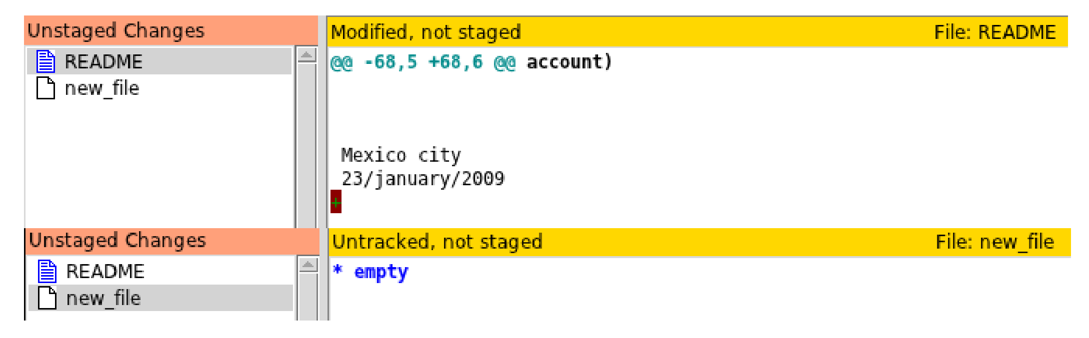
Inicialmente ningún cambio a ningún archivo sobre el que trabajemos, sea recién creado o sea modificado (aunque anteriormente ya hubieran sido versionados cambios al mismo) es considerado por Git para versionar. Es necesario hacer un `git add` para que Git sepa en particular qué archivos con cambios serán versionados.

Esto proporciona un control muy fino del proceso de versionado. En sistemas como SVN si un archivo es considerado para versionar, lo es desde que es agregado al repositorio y hasta siempre. Si deseamos modificar este archivo aún cuando esa modificación no tenga nada que ver con las modificaciones a otros archivos, el proceso de **commit** se llevará de una sola vez a todos los archivos versionados. En Git sin embargo tenemos la opción de elegir puntualmente qué archivos con cambios se van a versionar.

Así, si hacemos una modificación que sea una corrección de un bug en varios archivos, y a la vez modificamos otros archivos para corregir errores de tipografía en la documentación, Git nos permite versionar cada una de estas modificaciones por separado, permitiendo identificar más fácilmente qué archivos cambiaron como parte de qué modificación, sin confundir con otros archivos relacionados a otras modificaciones.

Por ejemplo:

```
$ touch new_file
$ echo "" >> README
$ git status -s
 M README
?? new_file
```



Hasta aquí se puede observar que, tanto al agregar un nuevo archivo, como al editar un archivo ya existente en el repositorio, Git sabe que ambos archivos no serían versionados en el próximo **commit**. Lo que sí sabe es que el archivo que ya había sido versionado ahora fue modificado, pero no por ello lo versionará de nuevo; y el archivo nuevo no sabe nada de él hasta ahora. Ahora:

```
$ git add README new_file
$ git status -s
M  README
A  new_file
```

Staged Changes (Will Commit)
README
new_file

Y ahora sí, luego de *git add*, Git sabe que ambos archivos deben ser versionados (uno por haber sido modificado y otro por haber sido añadido). Si sólo uno de los archivos debe ser versionado, entonces *git add* sólo recibiría ese archivo como argumento. Luego se versionaría con *git commit* y el otro archivo aún quedaría pendiente de versionar...

git status

Hasta ahora para demostrar *git add* se usó también *git status* sin explicar cómo funciona.

Básicamente, *git status* permite conocer la manera en que Git ve los archivos del proyecto con respecto al repositorio. La opción *-s* usada en los ejemplos anteriores dan un status resumido. Sin esta bandera la salida sería mas o menos así:

```
# On branch masterm lang
# Changes to be committed:
# (use "git reset HEAD ..." to unstage)
#
# modified: README
# new file: new_file
```

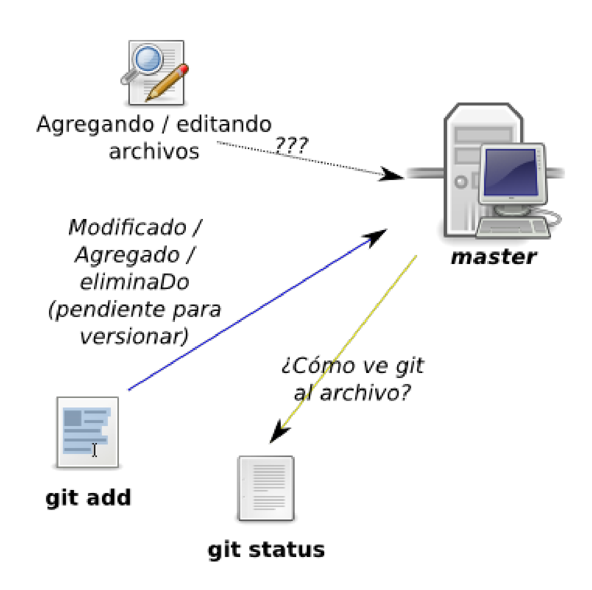
Si se puso atención, en la salida con la bandera *-s*, el status aparece con dos columnas. La primera indica los cambios que hará Git en la siguiente versión del código. La segunda columna indica cambios que Git reconoce como tales pero que no versionará:

```
$ echo "" >> new_file
$ git status -s
M  README
AM new_file
```

Como se puede ver aquí, luego de una modificación al archivo `new_file`, Git sabe que hay modificaciones, pero como fueron hechas luego de *git add*, Git no las tomará en cuenta para la siguiente versión. Observemos la salida no resumida del status:

```
$ git status
# On branch masterm lang
# Changes to be committed:
# (use "git reset HEAD ..." to unstage)
#
# modified: README
# new file: new_file
#
# Changes not staged for commit:
# (use "git add ..." to update what will be committed)
# (use "git checkout -- ..." to discard changes in working directory)
#
# modified: new_file
#
```

Es decir, en la siguiente versión aparecerán los cambios al archivo `README` y la aparición del nuevo archivo `new_file`, pero la modificación hecha a este último no aparecerá.

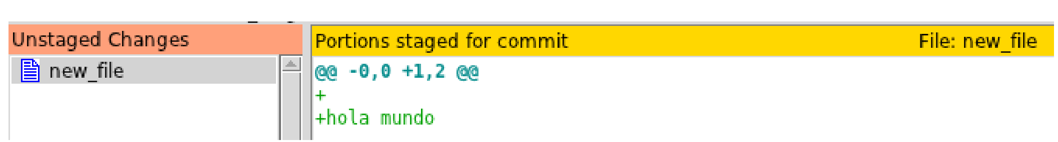


git diff

Otra operación muy común al trabajar con archivos versionados es la de observar y analizar los cambios hechos, así como las diferencias entre la nueva versión y la que se tiene en una versión anterior.

Git utiliza `diff` para esto mismo, como puede verse en el siguiente ejemplo:

```
$ echo "hola mundo" >> new_file
$ git diff
diff --git a/new file b/new file
index e69de29..775af59 100644
--- a/new file
+++ b/new file
@@ -0,0 +1,2 @@
+
+hola mundo
```



Esta salida indica que Git detecta cambios que no han sido marcados para versionarse en el archivo `new_file` (una línea nueva del ejemplo anterior, y un mensaje del presente ejemplo).

Si se deseara ver las diferencias que Git detecta tomando en cuenta los cambios en los archivos que ya fueron marcados para versionar, aunque aún no haya sido versionados, se utiliza el parámetro *--cached*:

```
$ git diff --cached
diff --git a/README b/README
index 058947a..082cf26 100644
--- a/README
+++ b/README
@@ -70,3 +70,4 @@ account)

Mexico city
23/january/2009
+
diff --git a/new file b/new_file
new file mode 100644
index 0000000..e69de29
```

Como puede observarse, aquí se muestran los cambios en el archivo `README` que ya habían sido marcados para versionar anteriormente (una línea nueva del primer ejemplo). También se muestra el único cambio hecho a `new_file` que ya había sido marcado para versionar: la creación del archivo nada más.

Por último, si lo que se desea es ver todos los cambios, tanto de lo marcado para verionar como lo que no, se le pasa a *git diff* como parámetro la versión contra la cual se quiere comparar el código actual. El nombre `HEAD` se refiere a justamente la última versión que tiene almacenada el repositorio:

```
$ git diff HEAD
diff --git a/README b/README
index 058947a..082cf26 100644
--- a/README
+++ b/README
@@ -70,3 +70,4 @@ account)

Mexico city
23/january/2009
+
diff --git a/new_file b/new_file
```

```
new file mode 100644
index 0000000..775af59
--- /dev/null
+++ b/new file
@@ -0,0 +1,2 @@
+
+hola mundo
```

Que como puede observarse, incluye los cambios tanto a `README` (previamente marcados para versionar) como los de `new_file` (algunos ya marcados y otros aún no marcados para versionar).

`git commit`

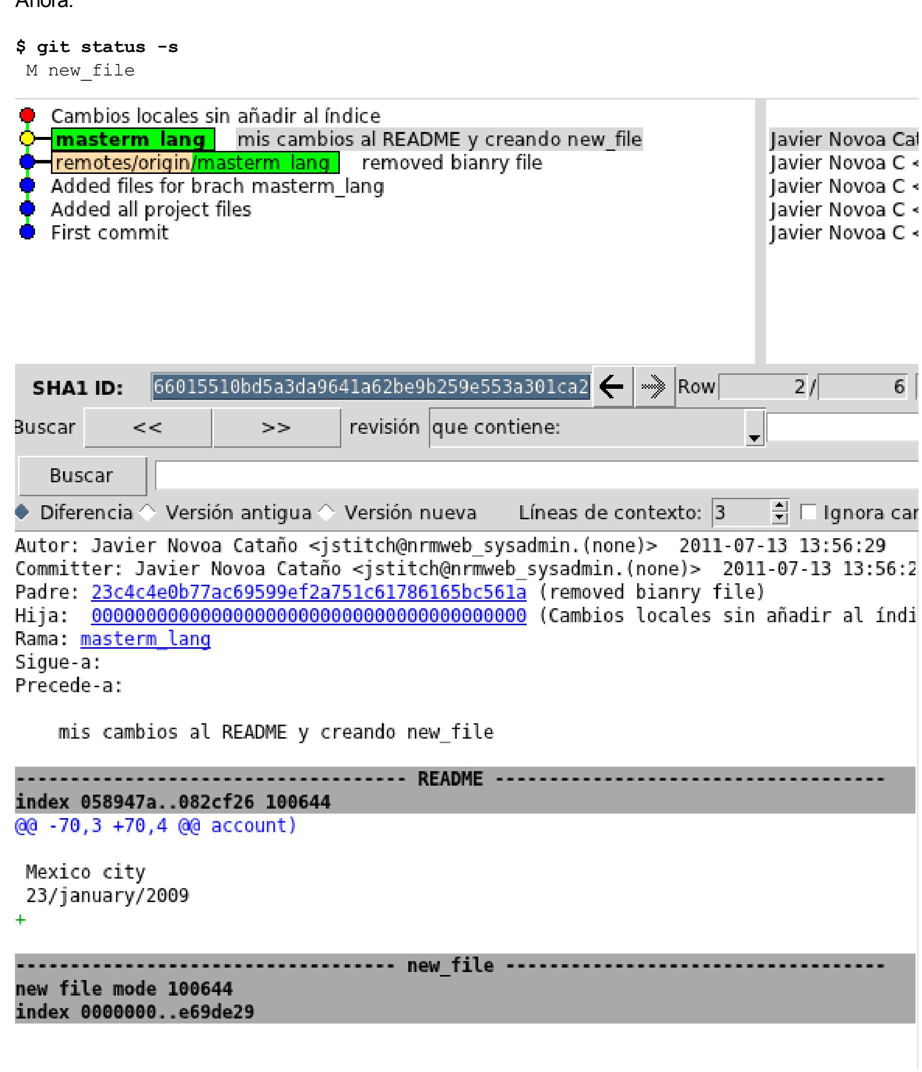
Finalmente, una vez hechos los cambios deseados, añadidos nuevos archivos, eliminados otros, y añadidos dichos cambios específicos para ser versionados por Git (ignorando por ahora los otros cambios que no deseamos versionar todavía), se realiza el **commit** al repositorio (recuérdese: es local, ¡no hay necesidad de conexión a la red todavía!):

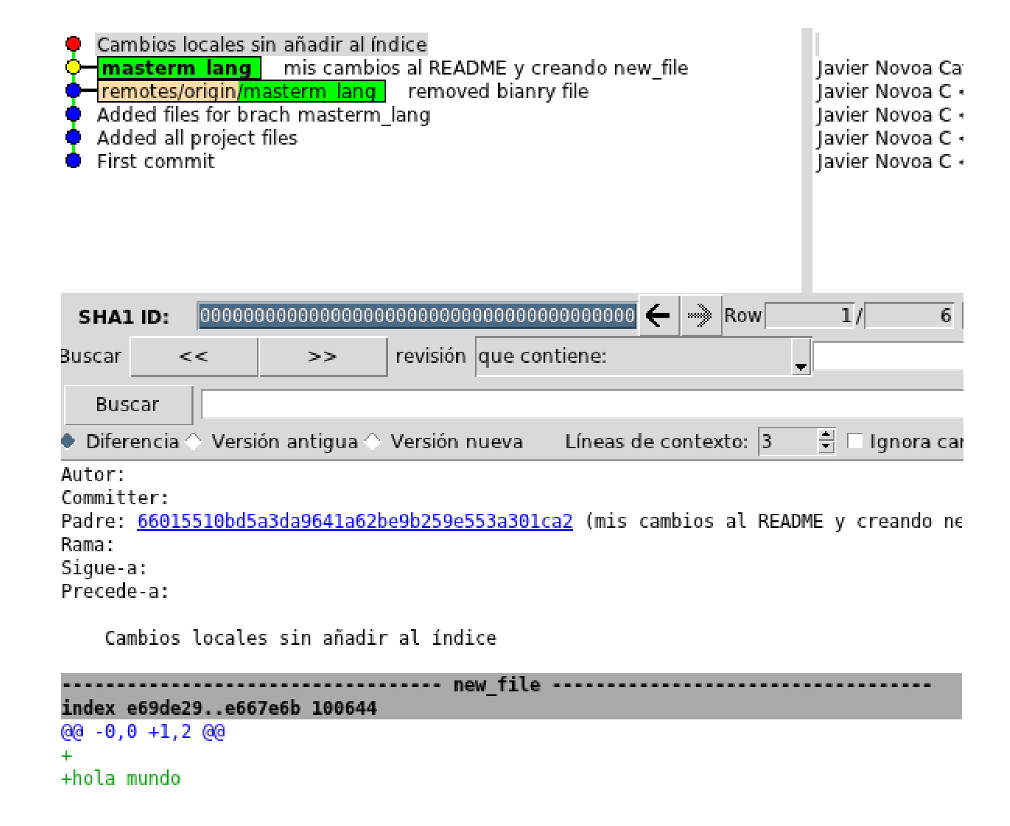
```
$ git commit -m "mis cambios al README y creando new_file"
```

Como se puede observar, el comando necesita de un mensaje descriptivo, y de hecho Git lanza error si no se agrega mensaje.

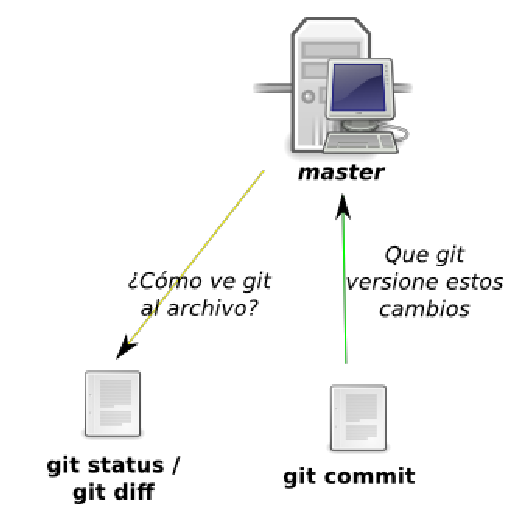
Ahora:

```
$ git status -s
 M new_file
```





Que nos muestra que sólo queda un cambio detectado por Git: aquel que no hemos marcado para versionar aún.



Branches y Merges

Para los provenientes de VCS centralizados como SVN, muy probablemente el nombre 'Branch' ya dibujo en su mente la idea de una pesadilla dantesca... La realidad es que el manejo de branches en versionadores centralizados suele convertirse en una molestia más que en una herramienta útil. Tan es así que muchos proyectos renuncian a su uso y se dedican únicamente a versionar sobre un árbol principal de código, dejando de lado una de las ventajas esenciales que proporciona un versionador.

Pero la buena noticia es que en los versionadores distribuidos, y en Git en particular, el manejo de branches es, dicho llanamente, *maravilloso*.

Una manera de entender los branches en Git es, además de olvidando lo aprendido en versionadores centralizados, verlos como *contextos* en los que se trabaja una versión específica del código.

Digamos que bajamos el código fuente de un software que utilizamos mucho y detectamos un bug. El modo Git de afrontarlo sería, luego de clonar el repositorio, crear un branch y ahí trabajar la corrección del bug. Si además resulta que tenemos una propuesta de mejora al software, lo correcto desde el punto de vista del modo Git de trabajarlo sería crear otro branch a partir del original (o *maestro*) y ahí trabajar los cambios. Finalmente cuando el bug esté corregido, se integran (vía **merge**) con el branch maestro. Y cuando nuestros cambios estén hechos y probados también, se hace lo mismo desde aquel otro branch al maestro de nuevo. Así, al final, se tiene un código limpio, probado, bien versionado. Todo gracias al uso de branches.

En resumen, se crean branches con *git branch*, se hace cambio de contextos con *git checkout* y se mezclan branches con *git merge*. Otra característica notable del manejo de branches de Git es que los branches creados no se crean aparte en subdirectorios dedicados a lo mismo, como en SVN. Más bien, el directorio .git contiene toda la información (en forma de snapshots o diferencias entre cada archivo y sus branches), y así en un sólo directorio de trabajo del proyecto, al cambiar de contexto, todo el código del mismo directorio pasa a ese nuevo estado. Se puede cambiar entre contextos libremente y sin pérdida de información, y sin el estorbo de un directorio dedicado a cada branch del proyecto.

git branch y *git checkout*

Como ya se insinuó más arriba. Al crear un nuevo repositorio en Git, por defecto se tiene un único branch, el inicial o principal del proyecto. Se le llama *master* por default. El comando *git branch* lista todos los branches existentes actualmente en un proyecto. Siguiendo con el ejemplo de un repositorio clonado previamente desde otro proyecto:

```
$ git branch
* masterm_lang
```

El asterisco muestra cuál es el branch actual (o contexto actual) en el que se encuentra el proyecto. Como puede verse, al tratarse de un proyecto clonado desde otro repositorio, el branch por default no se llama *master*, pero el punto es que se trata del branch principal del proyecto.

Para crear un nuevo branch, se utiliza el comando *git branch [nombre_del_branch]*:

```
$ git branch testing
$ git branch
* masterm_lang
  testing
```

Como se puede observar, ya existe un nuevo branch en el proyecto (*testing*). Este branch está hecho a partir del código en su último estado: tanto los últimos commits como los archivos con cambios que han y no han sido añadidos para versionar.

Y para pasar a ese nuevo contexto y trabajar sobre él, se utiliza *git checkout [nombre_del_branch]*:

```
$ git checkout testing
$ git branch
  masterm_lang
* testing
```

Y, como puede observarse, ahora es *testing* el contexto actual sobre el que se trabajará en el proyecto.

Si hacemos algunos cambios en un archivo, y luego regresamos al contexto original (*masterm_lang*), como no se le indicó a Git que los cambios se irían a versionar en ese contexto, los cambios pasan transparentes entre contextos:

```
$ echo "hola README" >> README
$ git checkout masterm_lang
M README
M new file
Switched to branch 'masterm lang'
Your branch is ahead of 'origin/masterm_lang' by 1 commit.
$ git status -s
 M README
 M new_file
```

Lo cual indica que los cambios que no se han marcado para versionar (recuérdese que *new_file* aún tiene cambios sin marcar) pasan entre contextos de manera transparente.

Si ahora marcamos algún cambio para versionar en uno de los contextos...:

```
$ git add new file
$ git status -s
 M README
M  new file
$ git checkout testing
M README
M new file
Switched to branch 'testing'
$ git status -s
 M README
M  new_file
```

Como se puede observar, aquí también los cambios marcados para versionar pasan transparentes entre contextos. Git no puede saber si la marca agregada para versionar es para uno u otro contexto, y sólo sabrá información más concreta hasta hacer un commit:

```
$ git commit -m "agrego texto a new_file, como prueba"
[testing 6870e92] agrego texto a new file, como prueba
 1 files changed, 2 insertions(+), 0 deletions(-)
```

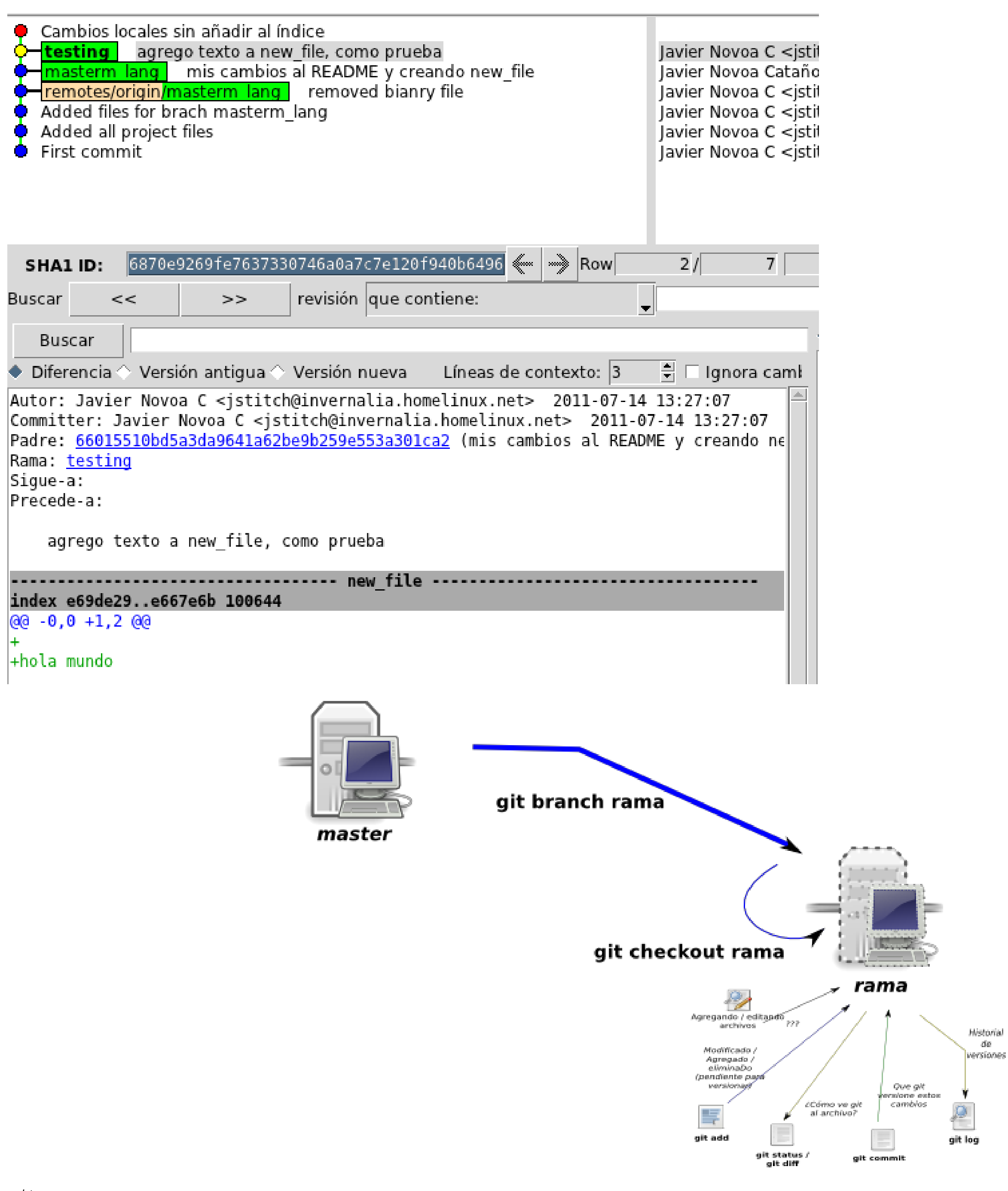
Y ahora sí, los cambios hechos a *new_file* quedan en el branch *testing*, no en la rama principal:

```
$ cat new_file

hola mundo
```

```
$ git checkout masterm_lang
M README
Switched to branch 'masterm_lang'
Your branch is ahead of 'origin/masterm_lang' by 1 commit.
$ cat new_file
```

En el branch *masterm_lang* no existe el cambio de la nueva línea y el texto "hola mundo". Pero en el branch *testing* sí que existen estos cambios pues ahí se hizo el **commit**.



git merge

Supongamos que ahora deseamos que el cambio en *testing* quede reflejado también en *masterm_lang*. Lo que debe hacerse es un **merge**, una operación que la mayoría de las veces Git puede hacer por su propia cuenta.

```
$ git branch
* masterm_lang
  testing
$ git merge testing
Updating 6601551..6870e92
Fast-forward
 new_file | 2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)
$ cat new_file
```

```
hola mundo
```

Conflictos

¿Pero qué pasaría si otro usuario hubiera hecho cambios a *new_file* sobre *masterm_lang* antes que nosotros hubiéramos hecho el **commit**? Entonces Git generaría lo que se conoce como un **conflicto**, que no es otra cosa sino la forma en que Git indica que no sabe como hacer el **merge** por sí solo y necesita de la ayuda externa de los usuarios. Los conflictos no ocurren siempre, incluso aunque muchos usuarios hagan cambios al mismo archivo muchas veces. Básicamente si los cambios se hacen sobre líneas diferentes del dicho archivo, Git sabrá hacer el **merge** sin problemas. Es cuando se hacen cambios que inmiscuyen líneas iguales en el archivo cuando Git puede verse en problemas... Veámoslo con un ejemplo, recordando que aún hay un cambio sin versionar en *README*:

```
$ git checkout testing
M README
Switched to branch 'testing'
$ git add README
$ git commit -m "agrego cambio a README de prueba, esperando generar conflicto en master"
[testing 6d32c82] agrego cambio a README de prueba, esperando generar conflicto en master
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Hasta aquí versionamos el cambio al branch *testing*...

```
$ git checkout masterm_lang
Switched to branch 'masterm_lang'
Your branch is ahead of 'origin/masterm_lang' by 2 commits.
$ echo "hello README" >> README
$ git add README
$ git commit -m "agrego cambio a README esperando generar conflicto cuando haga merge"
[masterm_lang aa7cca6] agrego cambio a README esperando generar conflicto cuando haga merge
 1 files changed, 1 insertions(+), 0 deletions(-)
```

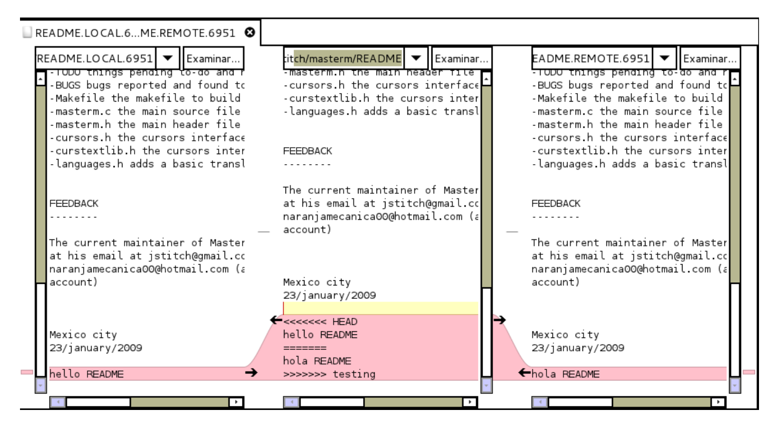
Ahora regresamos a *masterm_lang* y ahí hicimos un cambio diferente sobre el mismo archivo, en la misma línea (la última) que el cambio que se versionó en *testing*. Todo eso antes del **merge**. Y ahora a ver que pasa:

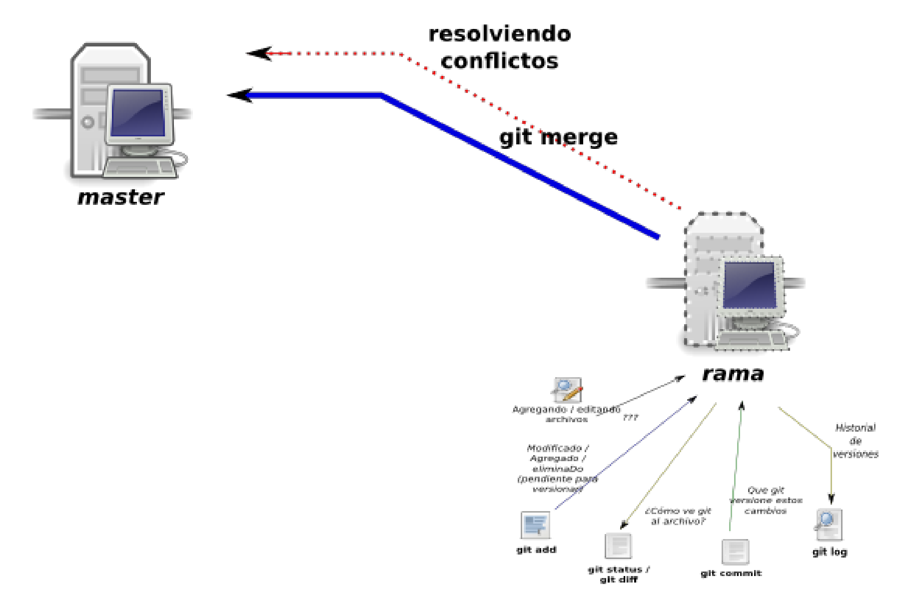
```
$ git merge testing
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
$ git status -s
UU README
$ tail README
Mexico city
23/january/2009

<<<<<<< HEAD
hello READM
=======
hola README
>>>>>>> testing
```

¿¿Qué sucedió?? Pues que Git no supo qué hacer y generó un conflicto al hacer el **merge**. Este conflicto queda marcado para Git (según el resultado de *`git status -s`*, y también dentro del mismo archivo README, como puede verse por los marcadores que se agregaron automáticamente al archivo en los lugares en donde ocurrió el conflicto.

Y para resolver el conflicto, se necesita la intervención humana. Normalmente aquí es donde los desarrolladores responsables de los cambios que ocasionaron el conflicto se ~~baten en duelo~~ ponen de acuerdo para decidir cómo resolver el conflicto. Al final, uno de los desarrolladores deberá editar el archivo con conflicto, dejar el cambio adecuado y retirar las marcas de conflicto (<<<< y >>>>) que colocó Git. Esto lo puede hacer editando manualmente el archivo o con alguna herramienta de resolución de conflictos, invocando *`git mergetool`*.





Y esa es la forma en que se trabaja con los branches en Git. Con otros versionadores es posible lograr este mismo tipo de proceso y organización, pero requiere de administración extra por parte del usuario, cosa a la que muchos proyectos no están acostumbrados. Obviamente, para lograr sacarle provecho a los branches, hay que también organizarse un poco. Este link puede ser útil para quien busque alguna guía práctica (en inglés).

`git log`

Un sistema que maneja tan eficientemente tanta información como Git no sería nada útil si no permitiera también mostrar de manera ordenada dicha información al usuario, de forma que él pueda saber con exactitud algún pedazo que le sea realmente útil. Para eso existe *`git log`*.

Este comando tiene en realidad muchos usos y muchas formas diferentes de generar y reportar la información con que cuenta Git, por lo que veremos solamente algunos ejemplos que podrían ser útiles:

```
$ git log
commit 8ca28c3c01257ec70816dee2d9a4f9338395ab04
Merge: 5d2d17e 2f77f01
Author: Javier Novoa C
Date: Fri Jul 15 10:38:12 2011 -0500
```

Merge luego de conflicto

commit 5d2d17e1773c417c89692db8e4eb7af46c442e13
Author: Javier Novoa C
Date: Fri Jul 15 10:36:07 2011 -0500

agrego cambio a README esperando generar conflicto cuando haga merge

commit 2f77f0110cc5138db58050ced9247beb51b8532d
Author: Javier Novoa C
Date: Fri Jul 15 10:35:37 2011 -0500

agrego cambio a README de prueba, esperando generar conflicto en master

commit 97bc9fbc9aa2e030c7981edafc9565f5a122f555
Author: Javier Novoa C
Date: Fri Jul 15 10:34:44 2011 -0500

agrego texto a new_file, como prueba

commit 4882d44687aa668fc4115f596552e431e5a6e9d1
Author: Javier Novoa C
Date: Fri Jul 15 10:32:33 2011 -0500

mis cambios al README y creando new_file

commit 23c4c4e0b77ac69599ef2a751c61786165bc561a
Author: Javier Novoa C
Date: Mon Mar 14 17:14:36 2011 -0600

removed bianry file

Esta es la salida por default de *`git log`*. Muestra en bloques cada uno de los commits que se han hecho dentro del branch actual (si estuviéramos en otro branch

y no se hubieran hecho los **merge** correspondientes, podría verse otra información también, aunque ambos branches coincidirían en su log desde el inicio de la existencia del branch padre desde el que se generó el actual hasta el momento en que se creó el branch nuevo).

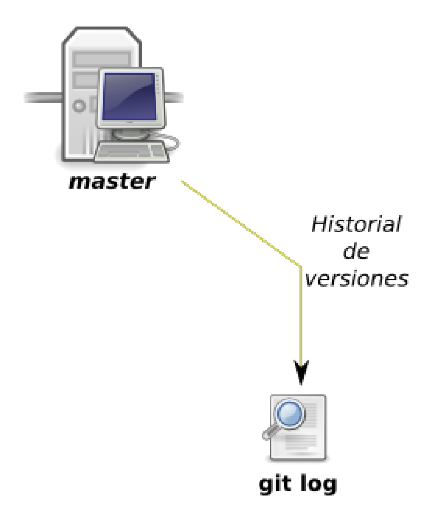
Se puede observar que se da información como una clave distinta para identificar cada **commit** hecho, los datos del responsable de tal **commit**, la descripción dada en su momento. E incluso cuando se trata de **merge**, se identifican los **commit** inmiscuidos. Este identificador de los **commit** se puede usar como parámetro a comandos como *`git diff`* para comparar el estado actual del proyecto con tal o cual versión del repositorio (en lugar del *HEAD*)

```
$ git log --oneline
8ca28c3 Merge luego de conflicto
5d2d17e agrego cambio a README esperando generar conflicto cuando haga merge
2f77f01 agrego cambio a README de prueba, esperando generar conflicto en master
97bc9fb agrego texto a new file, como prueba
4882d44 mis cambios al README y creando new_file
23c4c4e removed bianry file
```

Con el parámetro *`--oneline`* se muestra solamente el identificador corto del **commit** y la descripción del mismo, para un resumen breve. Nótese aquí la importancia de buenas descripciones en los **commit**: es la información de la evolución del sistema lo que se está describiendo, no hay que tomarse a la ligera escribir buenas descripciones...

```
$ git log --oneline --graph
*   8ca28c3 Merge luego de conflicto
|\
| * 2f77f01 agrego cambio a README de prueba, esperando generar conflicto en master
* | 5d2d17e agrego cambio a README esperando generar conflicto cuando haga merge
|/
* 97bc9fb agrego texto a new file, como prueba
* 4882d44 mis cambios al README y creando new_file
* 23c4c4e removed bianry file
```

Con el parámetro *`--graph`* se puede ver incluso de manera visual cómo han evolucionado los branches y los posibles **merge** hechos entre ellos. Una opción utilísima...



`git tag`

Para terminar con este asunto de los branches, vamos a mencionar los *tags*. Casi cualquier versionador permite manejar este concepto (unos más fácilmente que otros). La idea detrás de un *tag* es tener una especie de fotografía fija del proyecto en cierto momento. De esa forma, cuando se quiera tener el código del proyecto justo como se tuvo en el momento de tomar la 'fotografía', simplemente uno va al tag y lo recupera.

Esto es sumamente útil cuando, por ejemplo, se tiene el proyecto en un estado listo para liberar a un entorno de producción. Aún se planean mejoras, se planea mantenimiento, pero en ese estado justamente se decide tener la, digamos, versión 1.1.2 del software. Entonces se genera un tag del momento del proyecto deseado, se le etiqueta como 'versión 1.1.2' y listo! Git tiene la información que nosotros podemos usar cuando deseemos, por ejemplo regresamos el código al estado de ese tag y luego empaquetamos o compilamos o lo que fuera necesario...

```
$ git tag -a v1.1.2
$ git log --oneline --decorate --graph
*   8ca28c3 (HEAD, tag: v1.1.2, masterm_lang) Merge luego de conflicto
|\
| * 2f77f01 (testing) agrego cambio a README de prueba, esperando generar conflicto en master
* | 5d2d17e agrego cambio a README esperando generar conflicto cuando haga merge
|/
* 97bc9fb agrego texto a new file, como prueba
* 4882d44 mis cambios al README y creando new file
* 23c4c4e (origin/masterm_lang, origin/HEAD) removed bianry file
```

Como se puede observar, al usar el parámetro *`--decorate`* de *`git log`* (que muestra más información sobre los branches), también se muestra ahora el tag que acabamos de generar para el *HEAD* del repositorio. Obviamente, también se puede dar tag a alguna versión diferente al HEAD, para lo cual al comando *`git tag`* simplemente se le agregaría el identificador del commit al que deseemos ponerle el tag:

```
$ git tag -a v1.1.2-beta 97bc9fb
```

```
$ git log --oneline --decorate --graph
*   8ca28c3 (HEAD, tag: v1.1.2, masterm_lang) Merge luego de conflicto
|\
| * 2f77f01 (testing) agrego cambio a README de prueba, esperando generar conflicto en master
* | 5d2d17e agrego cambio a README esperando generar conflicto cuando haga merge
|/
* 97bc9fb (tag: v1.1.2-beta) agrego texto a new file, como prueba
* 4882d44 mis cambios al README y creando new file
* 23c4c4e (origin/masterm_lang, origin/HEAD) removed bianry file
```

Para pasar el código de un tag a otro, se puede usar el mismo comando *git checkout*, pero en lugar de usar el nombre de un branch como parámetro, se usa el nombre dado al tag (también se puede usar el identificador de un **commit** cualquiera). Solamente hay que tener cuidado, si el lugar al que nos movemos no es un branch específico, Git entra a un estado en el que la copia de trabajo actual no está en ningún branch, y es necesario moverse de ahí para continuar trabajando.

Por último, el parámetro *-l* informará de todos los tags que tenga creados el repositorio:

```
$ git tag -l
v1.1.2
v1.1.2-beta
```



Interactuando con un repositorio remoto

Como hasta ahora se ha podido constatar, Git no utiliza servidores centrales con los repositorios, a la manera de SVN y otros VCS. Un VCS distribuido como Git básicamente genera un nuevo 'servidor' Git por cada clonación de un repositorio Git que se haga, no hay ninguna diferencia entre el servidor y el cliente, ambos son el mismo y utilizan el mismo formato para la información que almacenan.

Una vez que se tiene un repositorio Git sobre el que versionar algún proyecto, se le puede indicar a Git que sincronice la información de este repositorio con algún otro repositorio remoto, de forma que este último pueda tener los últimos cambios que se han realizado sobre el repositorio local o que éste se actualice a los últimos cambios que otros han subido al repositorio remoto.

¡Y la ventaja con Git es que no hay necesidad de hacer esto de forma que coincida con ningún **commit**! La sincronización es un proceso totalmente diferenciado del versionado, y no interfiere con el funcionamiento mismo de versionar el código y manejar las versiones del mismo a conciencia.

En resumen, se utiliza *git fetch* para actualizar el repositorio local con cambios del remoto y *git push* para enviar los cambios (versiones hechas con **commit**) del local al repositorio remoto. La administración de repositorios remotos se logra con *git remote*.

git remote
Puesto que en Git un cliente y un servidor es básicamente lo mismo, se pueden tener declarados más de un repositorio remoto y elegir el que se desee usar en

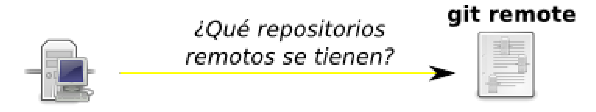
determinado momento, por ejemplo tener un repositorio de acceso completo y otro de sólo-lectura.

Al iniciar un repositorio con *`git init`* no hay ningún repositorio remoto declarado. Sin embargo con *`git clone`*, el repositorio remoto por default es el de la URL utilizada para clonar el repositorio. Por default el nombre de éste repositorio es *origin*. Git utiliza nombres cortos para identificar los repositorios remotos, de forma que no tenga que recordarse siempre, o estar tecleando, la URL de los mismos.

Con *`git remote`* se pueden ver los repositorios remotos que se tienen actualmente dados de alta:

```
$ git remote
origin
$ git remote -v
origin git://github.com/jstitch/masterm.git (fetch)
origin git://github.com/jstitch/masterm.git (push)
```

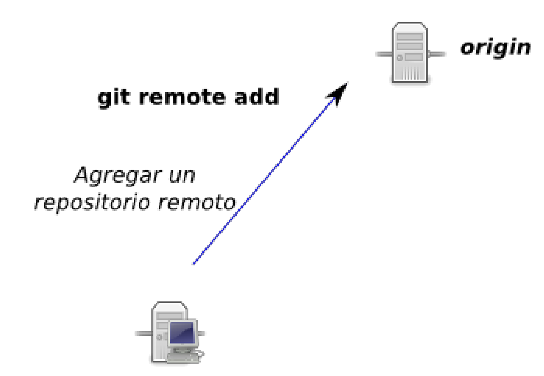
El parámetro *`-v`* muestra la URL asociada al remoto. Git incluso permite tener distintas URLs para un mismo remoto, dependiendo las operaciones que se quieran hacer sobre ellas: **fetch** (lecturas) y **push** (escrituras).



Con *`git remote add`* se puede dar de alta otro repositorio remoto:

```
$ git remote add invernalia git@invernalia.homelinux.net:masterm.git
$ git remote -v
invernalia git@invernalia.homelinux.net:masterm.git (fetch)
invernalia git@invernalia.homelinux.net:masterm.git (push)
origin git://github.com/jstitch/masterm.git (fetch)
origin git://github.com/jstitch/masterm.git (push)
```

Y como se ve, ya hay un nuevo repositorio remoto declarado en nuestro repositorio Git. Ahora, a usarlo... :)



`git fetch-merge / pull`

Con Git hay dos opciones para obtener los últimos **commit** de un repositorio remoto: *`git pull`* y *`git fetch`*/*`git merge`*. Básicamente, el **pull** hace un **fetch** seguido de un **merge**, pero si lo que se desea es tener mayor control sobre lo que se va a actualizar, puntualmente y procediendo branch por branch, el **fetch**/**merge** siempre es mucho mejor opción.

```
$ git fetch invernalia
From invernalia.homelinux.net:masterm
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From invernalia.homelinux.net:masterm
   23c4c4e..010040b  masterm lang -> invernalia/masterm lang
 * [new branch]      test remote -> invernalia/test_remote
 * [new tag]         v1.1.2-beta1 -> v1.1.2-beta1
```

Como se puede observar, en ese servidor remoto alguien ya hizo un **commit** sobre el branch *masterm_lang* y agregó un nuevo branch *test_remote* y un nuevo tag *v1.1.2-beta1*, cambios que ya fueron bajados aunque el código aún no refleja ningún cambio todavía...

Como puede observarse, Git hace un mapeo con los cambios hechos y los repositorios remotos. Así, el branch *masterm_lang* del remoto *invernalia* se llama ahora *invernalia/masterm_lang*. Ahora podría hacerse un *`git log`* para observar qué cambios serán sincronizados, y al final un *`git merge`*:

```
$ git log invernalia/masterm_lang --oneline --decorate --graph
* 010040b (tag: v1.1.2-beta1, invernalia/masterm lang) agrego un archivo en el remote
* 23c4c4e (origin/masterm_lang, origin/HEAD) removed bianry file

$ git log invernalia/test_remote --oneline --decorate --graph
* 4aa265d (invernalia/test remote) mas pruebas con archivo en branch de pruebas
* 010040b (tag: v1.1.2-beta1, invernalia/masterm lang) agrego un archivo en el remote
* 23c4c4e (origin/masterm_lang, origin/HEAD) removed bianry file
```

Es decir, hay un **commit** en *masterm_lang* y dos **commit** en *test_remote*, que por sus descripciones ya nos podemos hacer una idea de qué serán...

```
$ git merge invernalia/masterm_lang
Merge made by recursive.
 new file from remote | 1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 new_file_from_remote

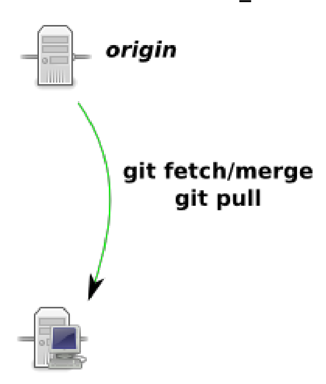
$ ls
BUGS curstextlib.h INSTALL languages.h Makefile masterm.h new_file_from_remote TODO
cursors.h HISTORY intl/ LICENSE masterm.c* new_file README utils.h
$ cat new_file_from_remote
hola desde el remote\!
$ git log --oneline --decorate --graph
*   c502d2e (HEAD, masterm_lang) Merge remote-tracking branch 'invernalia/masterm_lang' into masterm_lang
|\
| * 010040b (tag: v1.1.2-beta1, invernalia/masterm lang) agrego un archivo en el remote
* |   8ca28c3 (tag: v1.1.2) Merge luego de conflicto
|\ \
| * | 2f77f01 (testing) agrego cambio a README de prueba, esperando generar conflicto en master
* | | 5d2d17e agrego cambio a README esperando generar conflicto cuando haga merge
|/ /
* | 97bc9fb (tag: v1.1.2-beta) agrego texto a new file, como prueba
* | 4882d44 mis cambios al README y creando new_file
|/
* 23c4c4e (origin/masterm_lang, origin/HEAD) removed bianry file
```

Como puede observarse, se agrego un nuevo archivo, e incluso el árbol de versiones se actualizó con la información que se obtuvo desde el repositorio remoto...

```
$ git merge invernalia/test_remote
Merge made by recursive.
 new file from remote | 2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)
$ cat new_file_from_remote
hola desde el remote\!

linea 2
```

Y esto integra también a *masterm_lang* los cambios en el otro branch: el de *invernalia/test_remote*...



git push

En caso de tener permisos, un **push** nos permitirá subir nuestros cambios al repositorio remoto. De lo contrario, es necesario hacer una operación llamada **pull request** que explicaré más adelante.

```
$ git push invernalia masterm_lang
Counting objects: 22, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (18/18), 1.78 KiB, done.
Total 18 (delta 10), reused 0 (delta 0)
To git@invernalia.homelinux.net:masterm.git
   010040b..c932ba4 masterm_lang -> masterm_lang
```

Como se muestra, se enviaron al repositorio remoto los cambios del branch *masterm_lang*. También podrían enviarse los del branch *testing* que, dicho sea de paso, no existe en el repositorio remoto (eso créanmelo, así hice las pruebas ;):

```
$ git push invernalia testing
Total 0 (delta 0), reused 0 (delta 0)
To git@invernalia.homelinux.net:masterm.git
 * [new branch] testing -> testing
```

También enviamos los tags:

```
$ git push invernalia v1.1.2
Counting objects: 1, done.
Writing objects: 100% (1/1), 213 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@invernalia.homelinux.net:masterm.git
 * [new tag] v1.1.2 -> v1.1.2
$ git push invernalia v1.1.2-beta
```

```
Counting objects: 1, done.
Writing objects: 100% (1/1), 194 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@invernalia.homelinux.net:masterm.git
 * [new tag] v1.1.2-beta -> v1.1.2-beta
```

Date	Author	Message
11-07-15 18:47	Javier Novoa C	Merge remote-tracking branch 'invernalia/test_remo masterm_lang
11-07-15 18:44	Javier Novoa C	Merge remote-tracking branch 'invernalia/masterm_l
11-07-15 18:29	Javier Novoa C	mas pruebas con archivo en branch de pruebas test_remote
11-07-15 18:11	Javier Novoa C	agrego un archivo en el remote v1.1.2-beta1
11-07-15 15:38	Javier Novoa C	Merge luego de conflicto v1.1.2
11-07-15 15:36	Javier Novoa C	agrego cambio a README esperando generar conflicto
11-07-15 15:35	Javier Novoa C	agrego cambio a README de prueba, esperando genera testing
11-07-15 15:34	Javier Novoa C	agrego texto a new_file, como prueba v1.1.2-beta
11-07-15 15:32	Javier Novoa C	mis cambios al README y creando new_file
11-03-14 23:14	Javier Novoa C	removed bianry file
11-03-14 23:01	Javier Novoa C	Added files for brach masterm_lang
11-03-14 22:49	Javier Novoa C	Added all project files
11-03-14 22:47	Javier Novoa C	First commit

*Un vistazo al repositorio remoto luego del **push**: se ven los cambios que originalmente se habían hecho en el repositorio local*

git request-pull

Cuando no se tengan permisos para escribir en el repositorio remoto, pero aún así se desee intentar colaborar, Git proporciona un mecanismo para solicitar a quien mantenga el repositorio que le haga un **pull** (o un **fetch**/**merge**) a nuestro repositorio local y así lograr enviar nuestros cambios.

Para lograr esto, para empezar, el repositorio local debe estar configurado para poder proporcionar código a manera de un servidor, configuración que queda fuera del espectro de este artículo.

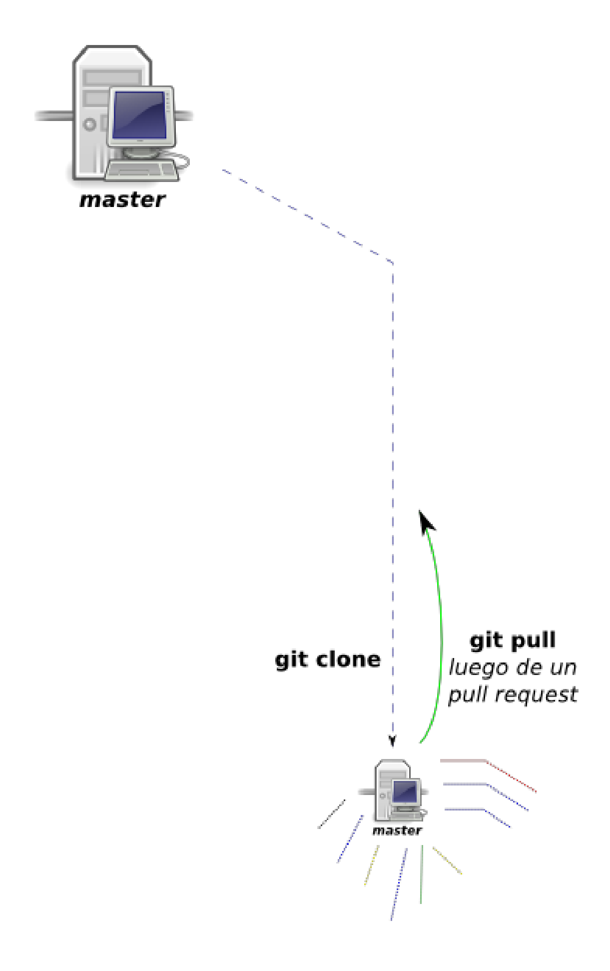
```
$ git checkout masterm_lang
Switched to branch 'masterm_lang'
Your branch is ahead of 'origin/masterm_lang' by 9 commits.
$ echo "nuevo cambio" >> README
$ git add README
$ git commit -m "otro cambio para probar pull request"
[masterm_lang 36260b0] otro cambio para probar pull request
1 files changed, 1 insertions(+), 0 deletions(-)
$ git tag v1.1.3-beta
$ git request-pull v1.1.3-beta git@invernalia.homelinux.net:masterm.git
The following changes since commit 36260b08b8c3e09baa1b4d882d82cc8620fdb016:
```

  otro cambio para probar pull request (2011-07-15 14:26:07 -0500)

are available in the git repository at:
  git@invernalia.homelinux.net:masterm.git/masterm_lang

Para más información, se puede consultar la página de manual de `git-request-pull(1)`.

v1.1.3-beta — masterm_lang — otro cambio para probar pull request
test_remote — remotes/invernalia/masterm_lang — Merge remote-tracking branch 'invernalia/test_remote' into masterm_lang
remotes/invernalia/test_remote — mas pruebas con archivo en branch de pruebas
Merge remote-tracking branch 'invernalia/masterm_lang' into masterm_lang
v1.1.2-beta1 — agrego un archivo en el remote
v1.1.2 — Merge luego de conflicto
testing — remotes/invernalia/testing — agrego cambio a README de prueba, esperando generar conflicto en master
agrego cambio a README esperando generar conflicto cuando haga merge
v1.1.2-beta — agrego texto a new_file, como prueba
mis cambios al README y creando new_file
remotes/origin/masterm_lang — removed bianry file
Added files for brach masterm_lang
remotes/origin/master — removed binary file
Added all project files
First commit



Conclusión

Con esto terminamos el recorrido por varias de las características más destacables de Git. No están todas las que son, pero al menos las que están pueden servir como un primer paso para comenzar a versionar utilizando esta grandiosa herramienta.

Git tiene muchas características y comandos más, a los que hay que ir revisando para utilizarlo en toda su potencia. Los comandos y opciones hasta aquí mencionados aún se pueden complementar con más opciones.

Y como siempre, para más referencias, consultar la documentación oficial, páginas del manual y google.

Lo único que queda ya es estudiar como utilizar el servicio GitHub, lo que haremos en la siguiente y última parte. Mientras tanto, como conclusión a todo esto, notar como el uso de un versionador se puede convertir en una de las herramientas más útiles para administrar y manejar código fuente en diversos proyectos. No es la única herramienta importante, mucho menos para proyectos de software libre, pero sí una de las más fundamentales. Claro, en algún momento también habrá que dedicarse a estudiar otras herramientas, en cuanto a administración: trackers de issues y bugs, posiblemente administradores de proyectos; en cuanto a programación: frameworks de pruebas unitarias y funcionales acordes al lenguaje en que esté hecho el proyecto, depurardores; en cuanto a documentación: frameworks para documentar código y generar documentación de APIs, wikis; y todo un largo etcétera...

Interfaces para Git

- git-gui sencilla interfaz gráfica portable basada en Tcl/Tk, es la interfaz gráfica 'oficial' del proyecto
- gitk interfaz gráfica para visualizar repositorios y sus historiales. También es parte 'oficial' del proyecto
- ViewGit un navegador de repositorios Git hecho en PHP
- TortoiseGit para Windows
- Tower para la Mac
- IDEs que tienen soporte para Git: Eclipse, Netbeans, Xcode

Para saber más...

Aquí pongo unos links con más información sobre Git:

- Manual de usuario de Git [en inglés]
- Tutorial de Git
- Referencia de Git [en inglés] - gran parte de la información de este artículo la base en éste
- gittutorial(7) [en inglés]
- A successful Git branching model [en inglés] excelente guía sobre cómo podría implantarse un modelo de versionado con branches usando Git en ambientes de desarrollo

---

parte 1 (Versionadores)
parte 3 (Github)

Añadir nuevo comentario

Utúlien aurë! Auta i lóme! Aurë entuluva!
The day has come! The night is passing! Day shall come again!
Fingon / Húrin Thalion
J.R.R.Tolkien, The Silmarillion